Chapter 9. The Future of Web Services

Throughout this book, we've maintained a fairly narrow focus on what web services are and how to go about creating them. To finish up, we will spend some time discussing the future of web services from both the point of view of the technologies, and from the point of view of the architecture as a whole. Specifically, we'll discuss the futures of SOAP, WSDL, and UDDI, as well as the next generation of even more useful and powerful services.

9.1 The Future of Web Development

Before we get into where the technologies are going in the future, let's take a moment to highlight exactly how web services are most likely to impact web development.

We've spent a great deal of time talking about interoperability. With web services, we're concerned about how to find information and move it across the Web. Issues that used to be important, such as the programming language in which the code was written, the operating system on which the code is running, the object model on which the code is based, or the vendor of the underlying database system, don't matter that much anymore.

Now that's a strong statement. Some may even say that it is too strong. Here's the reasoning.

First, prior to web services, the vast majority of enterprise-scale development platforms were rather inbred. Java applications worked best with Java applications; COM applications worked best with COM applications; CORBA applications worked best with CORBA applications, and so on. To make the most of each environment, you had to standardize and focus on the technology platform itself. You could get Java and COM to work together, but it was painful.

Web services, however, opened an integration channel between Java and COM, and COM and CORBA, etc., that did not exist before. Because this channel is built with open standards that any platform can implement, for the first time we have a situation where one can *easily* invoke functionality written in one programming language on one platform from any other programming language on any other platform. This allows us to look beyond programming languages and focus on the applications themselves.

This point was demonstrated in the Hello World example in Chapter 3. We created the same type of web service with three different programming languages. Barring minor interoperability bugs that exist in the web services tools, we could easily switch between the three service implementations without paying attention to how they were actually written. If we were to go looking for a Hello World service somewhere out on the Internet, it would be of no importance to me whether it is written in Java, Perl, .NET, or even COBOL or ADA. (There is an implementation of SOAP for ADA, by the way. And Microsoft's Visual Studio.NET will support writing assemblies with COBOL.)

9.1.1 Web Services and Existing Technologies

A critical insight is that web services *do not replace existing technology infrastructures*. Rather, they help to integrate existing technologies. In other words, if you need a J2EE

application to talk to a COM application, web services makes it easier. Web services won't completely replace that 30-year-old mainframe system in the back closet that nobody ever thinks about anymore. But web services might provide cross-platform automated access to the mainframe's applications, thus opening new channels of business.

9.2 The Future of SOAP

The SOAP protocol is already a couple of years old. One of the original versions became what is now called XML-RPC, a simple, popular alternative to SOAP championed by Userland Software. (Userland's CEO, Dave Winer, is one of the coauthors of the original SOAP specification.) To learn more about XML-RPC, read *Programming Web Services with XML-RPC*by Simon St. Laurent, Joe Johnston, and Edd Dumbill (O'Reilly).

XML-RPC split from SOAP in 1998. The first version of the SOAP protocol was announced in 1999, and since then there have been four revisions with a fifth now being worked on by the W3C. The two versions we've discussed in this book (Version 1.1 and Version 1.2) are those currently being used in production environments, even though they are not official W3C standards.

In the not-too-distant future, the SOAP 1.2 working draft specification will evolve into the W3C XML Protocol Version 1.0 recommendation, which will be the first *standardized* version of the protocol.

There should not be many changes between SOAP 1.2 and XML Protocol Version 1.0, because the W3C working group has committed to using SOAP as the basis for their work and to ensuring that backwards compatibility is maintained at least on a fundamental level. Unfortunately, it is still far too early in the process to know about any differences between the XML Protocol and SOAP 1.2. If you're curious, monitor the XML Protocol development discussion through the *xml-dist-app* mailing list (see the W3C XML Protocol home page at http://www.w3.org/2000/xp for subscription details).

9.3 The Future of WSDL

Like SOAP, the Web Service Description Language is not yet an official Internet standard, but it is well on the road to becoming one. It has been submitted to the W3C and a working group is being formed to manage it. Unlike SOAP, which has a fairly stable direction within the W3C, standardized WSDL may be different from the version being widely used in web services today. It's too early to know how different, or when the W3C-blessed standard will be released.

9.3.1 Missing Pieces

There are several important things missing from WSDL that will have to be addressed by the W3C working group. For example, there is no standardized mechanism for extending the WSDL description to include information about security requirements, quality of service attributes, sequencing of operations, and so on. While not a strict requirement when WSDL is used to describe simple, basic RPC-style services, such standard extensions become critical when applying web services technology to enterprise e-business scenarios.

9.3.2 An Alternative to WSDL

Another key issue that the WSDL working group will need to address is the reconciliation of WSDL to other, alternative service description mechanisms, such as the DARPA Agent Markup Language (DAML) based DAML-S (S stands for "services"). DAML-S is focused on the task of building a formal semantic data model for web services. In other words, they're formalizing the language we use to describe web services. While DAML-S has no solid corporate backing, much of the work being done will have an impact on the future direction of WSDL standardization.

The concepts involved in DAML-S are really not all that different from WSDL, but the syntax is much more complex. For instance, Example 9-1 shows a partial description of the Hello World service from Chapter 3, in DAML-S instead of WSDL.

Example 9-1. Sample DAML-S description of the WSDL service

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
         xmlns:service="http://www.daml.org/services/daml-
s/2001/05/Service#"
         xmlns:process="http://www.daml.org/services/daml-
s/2001/05/Process#"
              xmlns:profile="http://www.daml.org/services/daml-
s/2001/05/Profile#">

    <daml:Ontology about="">

        <daml:versionInfo>HelloWorld</daml:versionInfo>
        <daml:imports
              rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns" />
        <daml:imports
              rdf:resource="http://www.w3.org/2000/01/rdf-schema" />
        <daml:imports
              rdf:resource="http://www.w3.org/2000/10/XMLschema" />
        <daml:imports
              rdf:resource="http://www.daml.org/2001/03/daml+oil" />
        <daml:imports
              rdf:resource="http://www.daml.org/services/daml-
s/2001/05/Service" />
        <daml:imports
              rdf:resource="http://www.daml.org/services/daml-
s/2001/05/Process" />
        <daml:imports
              rdf:resource="http://www.daml.org/services/daml-
s/2001/05/Profile" />
    </daml:Ontology>

    <rdf:Service rdf:ID="StockQuoteService">
       <service:presents>
          <profile:Advertisement rdf:about="#StockQuote_Advertisement" />
       </service:presents>
       <service:implements>
          <process:ProcessModel rdf:about="#StockQuote_ProcessModel" />
       </service:implements>
    </rdf:Service>
```

```
    <process:ProcessModel rdf:ID="StockQuote_ProcessModel">
       <service:topLevelEvent rdf:resource="#GetStockQuote" />
    </process>

    <rdfs:Class rdf:ID="GetStockQuote">
       <rdfs:subClassOf
        rdf:resource="http://www.daml.org/services/daml-
s/2001/05/Process#Process" />
    </rdfs:Class>

    <rdf:Property rdf:id="symbol">
        <rdfs:domain rdf:resource="#GetStockQuote" />
        <rdfs:subPropertyOf
         rdf:resource="http://www.daml.org/services/daml-
s/2001/05/Profile#input" />
        <rdfs:range
            rdf:resource="http://www.w3.org/2000/10/XMLschema#string" />
    </rdf:Property>

    <rdf:Property rdf:id="value">
        <rdfs:domain rdf:resource="#GetStockQuote" />
        <rdfs:subPropertyOf
         rdf:resource="http://www.daml.org/services/daml-
s/2001/05/Profile#output" />
        <rdfs:range
            rdf:resource="http://www.w3.org/2001/10/XMLSchema#float" />
    </rdf:Property>

    <profile:Advertisement rdf:ID="StockQuote_Advertisement">
       <profile:serviceName>StockQuoteService</profile:serviceName>
       <!-- elements removed for brevity -->
    </profile:Advertisement>
</rdf:RDF>
```

DAML-S is based on the Resource Description Framework (RDF) standard. This tends to make it more complex, verbose, and difficult to use than WSDL.

That said, there are several lessons WSDL can learn from DAML-S:

1. DAML-S naturally supports the ability to extend service descriptions to include a wide variety of semantic and functional information such as security, quality of service, etc.
2. DAML-S naturally supports inheritance throughout the entire description.
3. DAML-S provides a rich mechanism for describing web service processes (logical sequences of operations). WSDL does not support sequencing operations at all.
4. DAML-S allows a service to implement multiple processes (the DAML-S equivalent to a WSDL port type).
5. DAML-S supports a rich service advertisement description that provides information about who is providing the service, what the provider's capabilities are, and so on. WSDL does not include any advertisement information at all.

These features of DAML-S are likely to be a part of the next generation WSDL.

9.3.3 Standard Extensions

One key component of the success of web services will be the ability to describe not only the service itself, but also all of the services' capabilities, requirements, assumptions, and processes in a standard, consistent way.

For example, consider how to describe a web service that uses a SAML-based single sign-on like the one we discussed in Chapter 5. There is no way to declare the type of authentication a service supports in WSDL. For that matter, there is no way to declare that a service supports any type of authentication.

How might the WSDL of the future let you express what type of authentication mechanism is used? One way would be to define a standard extension to the WSDL binding element, as in Example 9-2.

Example 9-2. Hypothetical extension to WSDL bindings

```
<binding name="HelloWorldBinding" type="HelloWorldPortType">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
   <s:authentication method="http://schemas.xmlsoap.org/security/saml" />
</binding>
```

WSDL-enabled web services tools that understand the authentication extension would then know that SAML could be used for authentication.

Currently, though, there are no standard extensions to WSDL, nor is there a broad industry effort to define them. This also may become part of the W3C-standardized WSDL.

9.4 The Future of UDDI

Chapter 4 introduced UDDI as a web service for discovering other web services. Through the definition of a standardized registry format and port type interface, UDDI allows service providers and service consumers to dynamically discover and integrate with one another.

UDDI was originally developed by Microsoft, IBM, and Ariba, and is now managed by a broad industry consortium of companies. The plan is to submit UDDI for standardization once Version 3.0 of the specification is finished (the current version is 2.0).

One of the key requirements for future versions of UDDI is a security infrastructure that would allow service consumers to validate the identity of service providers publishing their services—allowing for a much more robust trust relationship to be established.

9.4.1 Problems with UDDI

There are some problems with UDDI that will need to be addressed in future versions of the specification. Weak security is one of the most significant issues. Currently, it is possible for anybody to create an entry in a UDDI registry, pretending to be somebody else. For example, I can easily create an entry in a UDDI registry pretending to be Microsoft. Needless to say, this is not good.

Another major problem is the proliferation of "bad links" in public UDDI registries. These links point to companies or services that don't exist or are no longer available.

There is a lack of understanding in companies about what UDDI is for and how it may be useful. This might hamper its adoption in future. Even among companies who understand it, there have been some doubts raised about whether public UDDI registries will be useful in the long term.

9.5 Web Services Battlegrounds

Over the last few decades, we've seen companies go to war to establish their operating systems, component models, programming languages, browsers, and so on. One refreshing aspect of the web services world is that most of these battles become irrelevant. Consider the SOAP services and clients we've discussed in this book. When we deploy a SOAP service, we define the methods we want to expose across the network. In the past, we'd have defined those methods with CORBA IDL or something similar, generated language bindings for various programming languages and platforms, then hoped we could get enough of the marketplace to use our service. If your platform or your development tools weren't compatible with your infrastructure (maybe they didn't support the correct level of CORBA, for example), you would probably be out of luck.

With SOAP, we can describe everything in terms of platform-independent XML Schema data types. If your development platform has XML parsing tools (and these days you're hard-pressed to find a platform that doesn't, from mobile phones to mainframes), you can start developing applications that use the service.

Don't think for a moment that the fierce competitors of today's marketplace will suddenly get along swimmingly, though. As companies discover that the old battles no longer matter, everyone will try to get an edge on their competitors in some other way. We'll take a look at a couple of the battlegrounds of the future.

9.5.1 Development Tools

One of the reasons for the dominance of the Windows platforms is Microsoft's success in courting developers. Whatever the benefits of your technology, if you can convince hundreds of thousands of clever people to start building products with them, you gain an overwhelming advantage in the market. You don't have to come up with the killer app yourself; third-party developers can do that for you.

With that in mind, you'll see the major software vendors working very aggressively to differentiate their web services development tools. If I can convince you that my tools will make you infinitely more productive and successful, the task of locking you into my development tools becomes much easier. And once you're comfortable with my development tools, I can integrate my proprietary technology initiatives with those tools, slowly removing your ability to use other tools.

Vendors must appear to be standards-compliant, yet also seem somehow superior. A lot of the differentiating features will be nonstandard add-ons, a form of "embracing and extending" that has the potential to weaken web services interoperability while locking in developers to one vendor's products.

9.5.2 Killer Services

If millions of developers can access web services with free tools, one obvious business model is to provide web services so cool that developers will be willing to tie themselves to those services. This is similar to the Web, in which millions of customers can access web sites with a free browser.

One early contestant in the race for killer services is the online wallet. A next-generation online wallet is a web service that allows customers to store passwords, credit card numbers, and other sensitive information. The online wallet provider becomes a clearinghouse for e-commerce. If we want to set up an online store, we can use the service to process credit card transactions. A customer gives us some information (username and password, for example), and we access the online clearinghouse to get an approval code for the transaction.

We might have to pay the clearinghouse a fee (a percentage of the total, perhaps) for each transaction, but if this service is easy to use and access, provides a high level of service, is secure, and is widely accepted by consumers, we could save ourselves a great deal of time and headache in operating and managing our online store. If development tools make it very easy to use a particular online wallet, the vendor behind the development tools and the online wallet is in a very good position. This is widely thought to be part of Microsoft's .NET and .Net My Services (formerly known as Hailstorm) strategies.

As web services take hold in the marketplace, we'll see lots of providers try to come up with other killer services to bring the world to their online doorsteps.

9.5.3 Lucrative Marketplaces

The EDI industry has worked for decades to automate the exchange of purchase orders, invoices, and similar documents. Unfortunately, these systems have traditionally been very expensive to create and maintain. With the lower startup costs of web services (you can build, deploy, and access web services with the technologies you already have), many smaller firms can now participate in these online business communities, just as the advent of the Web introduced many new companies that gave established merchants a run for their money.

As the web services revolution takes off, we'll see the industry try once again to establish business-to-business (B2B) marketplaces. In the past, these have failed for two reasons:

- Buyers wanted more control over their buying decisions; they didn't want a machine to make a buy decision based on which company came up first on the alphabetized list of search results.
- Providers wanted more control over pricing. A marketplace in which a seller's prices are shopped around online might be good for an agent trying to find the lowest online price, but it's not good for the providers, particularly when an agent might not take into account such things as a provider's ability to handle large orders, how other buyers have rated a particular provider, etc.

As web services mature, these concerns will be addressed. Through SOAP method calls to a UDDI registry, an online buyer can find all of the providers that claim to meet the buyer's needs. New web services built on top of UDDI will allow agents to get more information about providers, including their credit ratings, how quickly they've delivered orders in the

past, etc. Other services can reassure providers that buyers will be able to compare different providers fairly. For example, my company may have slightly higher prices, but we don't claim to have products in stock when our warehouses are empty.

Web services promise to create an environment in which agents can evaluate various factors the way a human would, allowing those human users to focus on things more important to their businesses.

9.5.4 The Enterprise

Perhaps one of the most significant battles yet to emerge will be the one for dominance in the market for enterprise web services. These are the infrastructure services that will provide the foundation for delivering on the promise of agents, and more dynamic forms of e-business. These services include such things as distributed trust management and negotiation; metering, accounting, and billing; content and information management; privacy enforcement and auditing; intelligent and dynamic sourcing and materials procurement; and any number of other services that provide the bedrock of enterprise business development. It is still unclear what effect basing such core pieces of the infrastructure on web services technology will have on the marketplace, and at this point, far too early to offer any real insight. Whatever the impact, expect to see much more activity in this area in the very near future, as Internet technology companies (both old and new) vie for position in a burgeoning new market.

Web services are a young approach to writing distributed applications. As such, they are nowhere near as mature and feature-rich as mechanisms like J2EE, CORBA, and .NET. Particularly needed is functionality that enables web services to operate in the enterprise environment: security, transactions, database integration, etc. This is similar to the early days of Java—it took until Java 2 Enterprise Edition for programmers to have a set of standard extensions to Java for security, transactions, messaging, server support, databases, etc.

With web services, we see a parallel evolution. Currently, we have the technologies (e.g., SOAP, WSDL, and UDDI) for allowing web services to function. By themselves, these technologies hold great promise, but they are not quite enough for the enterprise environment.

9.6 Technologies

Although many web services standards are already defined, there are also many technologies that aren't quite there yet. We'll discuss those missing pieces, and speculate about how and when those missing pieces will be filled in.

9.6.1 Agents

An agent is a program that can act on your behalf. For example, I'd like to have an agent make flight, rental car, and hotel reservations for an upcoming business trip. My ideal agent would know which airlines and hotels I prefer, possibly based on previous trips to the same region. If we assume that all of the relevant data our agent might use is in a richly structured XML document, an agent might be programmed to take advantage of all sorts of information when planning a trip. For example, when flying coast to coast, Chicago is more likely to have weather delays in the winter, while Dallas is more likely to have weather delays in the summer. An agent could find out that there is a frequent-flyer promotion that would give me

10,000 extra frequent-flyer miles if I fly through Toronto. Maybe an agent could automatically check my calendar to see what time I'm free to leave the day of my flight.

Agents have been an AI pipedream for years. XML and web services have the potential to make them real, though. Here's what's needed:

- All the data involved must be encoded in XML, using well-understood vocabularies. That means we need standard tag sets for calendars, flights, airports, weather forecasts, etc. A few of those vocabularies exist, but most of them will need to be created.
- All of the various airlines, hotels, rental car companies, and other vendors must provide web services that make it easy for my agent to create, change, and cancel reservations.
- Most importantly, the agent technology must be powerful, reliable, secure, and easy to use. That's not exactly the easiest task in the world of software development. People won't use agents if they are untrustworthy, can't do much, or are too complicated for anyone without a Ph.D. in Computer Science.

9.6.2 Quality of Service

Web services make it possible to build applications from multiple components spread out across the Web. That's a very powerful notion, but for some applications, developers need assurance that those components will be available constantly with acceptable speeds. That means Quality of Service contracts will become even more important, simply because the Web will become a vital part of more and more applications.

9.6.3 Privacy

If the devices and agents in my life have been entrusted with sensitive personal data, it's crucial that they understand my wishes about privacy. It's also crucial that those devices and agents understand how various entities around the network will handle that data.

The Platform for Privacy Preferences (P3P) work done by the W3C will become increasingly important. P3P documents are machine readable, meaning that agents and other pieces of code can examine a site's privacy policy and determine whether it is acceptable.

As the importance of privacy grows (as well as the public's awareness of how little the Web actually has), other privacy technologies may be needed. For example, an agent could get a digitally signed and encrypted P3P document from a provider, obtaining a legally binding agreement that data supplied to the provider by the agent will be protected and handled a particular way.

The first step is relatively simple: create a P3P policy and associate it with your web service through links provided in the WSDL description of that service. However, this is only part of the solution. What is needed is a more comprehensive, *standardized* infrastructure for protecting information as it travels across the Web. Until such a framework is in place, the impact and usefulness of web services geared at handling personal information will be limited at best. Currently, there are no proposals on the table for doing this.

Example 9-3 shows what a P3P policy reference might look like from within a WSDL document. Here, we are stating that the *Privacy.xml* P3P policy applies to every operation defined by the `HelloWorldBinding`.

Example 9-3. P3P within WSDL

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/">
   <binding name="HelloWorldBinding" type="HelloWorldPortType">
      <P3P:POLICY-REFERENCES>
         <P3P:POLICY-REF about="Privacy.xml">
            <INCLUDE>*</INCLUDE>
         </P3P:POLICY-REF>
      </P3P:POLICY-REFERENCES>
   </binding>
</definitions>
```

9.6.4 Security

Beyond everything else, security is paramount. It doesn't matter what a given web service can do—if it's likely to give away my credit card number, I don't want to use it. Although the base SOAP specification itself was not designed with security in mind, that doesn't mean security is impossible.

One of the examples we've discussed in this book uses IBM's XML Security Suite to encrypt the contents of SOAP envelopes as they move across a network. As web services take hold, we'll see more technologies like this, with the end result being that secure SOAP envelopes will become as common as HTML documents transmitted across the Secure Socket Layer.

The question of security demands a complex answer—one that always comes back around to point not at the technology, but at how that technology is implemented, deployed, and used. Technology companies can only do so much in the way of providing methods of expressing trust or asserting facts. Security happens only when businesses take the time to make it a priority.

9.6.5 Trust Management

Trust is the paramount requirement for conducting business over the Internet and will be a key component to the success of the web services architecture. Already technologies are emerging that help companies express and establish trust relationships within the context of web services. One example of such a technology is the XML Key Management Service, a standard mechanism for managing public and private keys.

9.6.6 Online Contracts

We've talked about contracts and other legally binding documents throughout this section, emphasizing the point that if web services are commonplace, the impact of a particular service being unavailable or providing incorrect data could be catastrophic. How will those contracts be negotiated or enforced? Clearly, having the attorneys for the service provider meet with the attorneys for the service requestor won't work in a world of applications built from conglomerations of services.

Several attempts have been made to create XML-based languages capable of describing agreements and contracts. The Collaboration Profile Protocol and Collaboration Profile Agreement (CPP-CPA) from ebXML is one such technology. Unfortunately, none of these attempts have been widely adopted and the ultimate winner is yet to emerge.

9.6.7 Reliable Messaging

Reliable messaging involves ensuring that both the sender and recipient of a message know whether a message was actually sent and received, and ensuring that the message was sent once and only once to the intended recipient. It is a problem that has plagued Internet application development since its inception.

The Internet is, by its very nature, unreliable. Servers that were up and running one moment may be down the next. The protocols used to connect senders and receivers have not been designed to support reliable messaging constructs, such as message identifiers and acknowledgments. Recipients of messages must be able to acknowledge that they did in fact receive a message. Senders of messages must be able to cache those messages in the event that an acknowledgment is not received and the message needs to be sent again. The fundamental technology that drives the Internet today does not support such mechanisms. Therefore, we are forced to implement new protocols and technologies that address these needs.

The importance of reliable messaging within the enterprise cannot be understated, especially when we are discussing the implementation of web services that may span across firewalls to integrate with customers, suppliers, and partners.

Within the enterprise, reliable messaging has typically been provided by proprietary solutions such as IBM's MQ Series or Microsoft Message Queue, neither of which are capable of integrating easily with each other (there are ways to make them work together, but they are painful at best).

From the context of web services, there are two ways to approach the implementation of reliable messaging:

1. You can implement reliable messaging on the application layer, meaning that the tenets of reliable messaging must be incorporated directly into the implementation of the web service.
2. You can implement reliable messaging on the transport layer, meaning that web services don't have to do anything to support the use of reliable messaging.

The first approach is implemented by products such as Microsoft's BizTalk, which uses web services technologies such as SOAP to exchange business documents (e.g., purchase orders and requests for quotes) in a reliable way.

The second approach is implemented by protocols such as IBM's Reliable HTTP (HTTP-R). HTTP-R is an implementation of standard HTTP with the addition of "endpoint managers" that ensure the reliability of the connection between the HTTP requester and the HTTP server.

A full discussion of HTTP-R and BizTalk are out of the scope of this discussion. For more information on them, see the online references in Appendix A.

9.6.8 Transactions

One of the key requirements for applications deployed within an enterprise is the support of transactions. Multiple operations that need to be executed in a batch must either all succeed or all fail in order for any of the operations to be valid. Currently, there is no standard (or even proposed) method for implementing and managing transactions in the web service environment.

There is a long-running debate as to whether web services require a method for doing two-phase commit style transactions. A two-phase commit transaction is one in which all of the operations in a batch must be invoked, but not finalized. Once all operations report successful invocation, they may all go back and finalize their operations. The classic example of a two-phase commit is when an application needs to write data to two different tables in a database. Both tables must be updated or neither of the tables can be updated. If the `write` operation on one table succeeds, but the `write` operation on the second table fails, the first `write` must be undone and an error reported back to the user.

The primary problem with two-phase commit on the Web is that when each of the participants in the transaction (for example, the two database tables in the previous example), is waiting for the final confirmation that all of the operations have been completed successfully, they must hold a lock on the resource being modified within the transaction. This lock prevents anybody else from making changes to the resource that otherwise may have caused the transactions to fail. These locks are fine when all of the resources are being managed by the same computer, but cause performance, scalability, and reliability problems in a distributed computing environment.

This problem goes back to the discussion of reliable messaging. With web services, by far the most amount of traffic will be over HTTP. Without the promise of absolute reliability, if the connection between two participants in a transaction is broken while the transaction is being carried out, neither participant can finalize their operations because neither can figure out if the other's operation completed successfully. The locks placed on the resources in question could be held indefinitely, and processing would grind to a halt.

One promising IBM research project in the transaction area is something called a *Dependency Sphere*. A Dependency Sphere, or D-Sphere for short, is a new way of looking at transactions from a distributed computing, messaging-based viewpoint. In a two-phase commit, a transaction is successful if all of the operations executed within the context of that transaction perform without generating any errors. In the D-Sphere approach, the transaction is successful if all messages sent are reliably received and acknowledged by the intended recipient of those messages.

D-Spheres applied to web services introduce a new type of web service for managing the D-Sphere transaction context. It is the job of this management service to ensure that the transaction either succeeds or fails. If it fails, a notice will be sent to the participants of the transaction so that they can make the appropriate compensating actions. The advantage to this approach is that reliable messaging is assumed (so temporary disconnections between participants are no longer a factor) and resource locks are not necessary, stopping the types of deadlocks that could occur with a two-phase commit approach.

An example of how D-Spheres might come into play within an enterprise web services environment is when a service requester must perform multiple operations on multiple services—for instance, creating a new user in CRM and ERP services at the same time. The D-Sphere could ensure that both services successfully receive and acknowledge the request to add a new user. Appendix A has pointers to more information on D-Spheres.

9.6.9 Licensing and Accounting Services

Part of the web services vision is the idea that software can be sold as a service. That is, companies will pay to lease access to applications rather than take on the cost of purchasing and maintaining the applications themselves. This concept can ease maintenance costs, but requires standard web services for managing licenses and monitoring the use of services.

Within the enterprise, these services will have to integrate with existing accounting and billing solutions, authentication and authorization solutions, and event and notification services in order to be meaningful and useful.

9.7 Web Services Rollout

How are web services likely to be rolled out in the marketplace? We think the most likely scenario is that customers will build web services internally, then move on to applications built with more broadly distributed web services.

We've already discussed the technologies that must be built on top of SOAP and related technologies for web services to bear more of the weight of business. Given that issues like security, authentication, and nonrepudiation are difficult to address on the Web of today, we feel that many early adopters will start by implementing web services internally. As a network administrator, I can control access to internal servers much more easily than I can control access to a public web site.

As an example, say I build a SOAP-based application for processing expense accounts. Whenever a user returns from a business trip, she uses the SOAP client application to fill out her expense report. The SOAP client sends a query to the local UDDI registry, which points the client to a WSDL document, which provides the information the client needs to access the expense account application. The head of the accounting department can move the location of the expense account application at any time, and the client will still be able to find it and access it.

Because the application is built on SOAP, it's possible (it might even be easy) to write client applications that work on almost any platform I support. Because all the clients are internal to my network, I'm less concerned about security and privacy than I would be otherwise. Because the metadata about the application is described with WSDL and stored in a UDDI registry, I can change the location, host platform, host language, etc., of the application without affecting the clients. This gives system administrators a tremendous amount of flexibility.

As more and more internal applications are built with web services, we'll see early adopters start to bring in their vendors and business partners. It's great that I can do an inter-company requisition for supplies; the obvious next step is to do requisitions from outside suppliers. That next step requires that my suppliers use SOAP (and WSDL and UDDI and . . .) as well.

Applications based on web services will become commonplace, and a component architecture based on SOAP will become the dominant development paradigm.